CS 70 Discrete Mathematics for CS
Spring 2005 Clancy/Wagner **Notes 5**

Divide-and-Conquer and Mergesort

One of the most basic tasks performed by computers is *sorting*: i.e., given n items from a totally ordered set, put the items into ascending order. You can think of the items as English words, and the order as alphabetical order: given n English words, sorting requires that we put these words into alphabetical order. To avoid tiresome details, we will assume throughout this lecture that the n items to be sorted are all distinct.

How can we devise a general procedure for doing this? After a little thought, most people usually come up with something like one of the following:

Method 1 By scanning through the list once, find the largest item and place this at the end of the output list. Then scan the remaining items again to find the second largest, and so on. (This method is generally known as **selection sort**.)

SELECTION SORT

Method 2 Take the first item and put it in the output list. Then take the second item and insert it in the correct order with respect to the first item. Continue in this way, each time inserting the next item in the correct position among the previously inserted items—this position can be found by a linear scan through these items. (This method is known as **insertion sort**.)

INSERTION SORT

How good are these methods? Well, it's not too hard to see that they are both *correct*, i.e., they both result in a correctly sorted version of the original list. [As an exercise, you might like to state this fact formally and prove it by induction on n, for each method.] But how efficient are the methods? Let's look at selection sort first. It's easy to see that the first scan takes exactly $n-1$ item comparisons to find the largest element; similarly, the second scan takes $n-2$ comparisons; and so on. The total number of comparisons is thus

$$(n-1)+(n-2)+\ldots+2+1=\sum_{i=1}^{n-1} i = \tfrac{1}{2}n(n-1),$$

(where we have used a formula for the sum that we proved by induction in Lecture Notes 1).

What about insertion sort? Well, to insert the second item requires one comparison; to insert the third item requires (in the worst case) two comparisons; and in general, to insert the ith item requires (in the worst case, where we have to scan the whole list) $i-1$ comparisons. Thus the number of comparisons used by the entire procedure in the worst case is

$$\sum_{i=1}^{n-1} i = \tfrac{1}{2}n(n-1),$$

exactly the same as for selection sort.

Thus the number of comparisons performed by both methods is at most $\frac{1}{2}n^2 - \frac{1}{2}n \approx \frac{1}{2}n^2$ for large n. Since comparisons constitute the bulk of the work performed by the algorithm, we can think of n^2 as a measure of the efficiency of the algorithms (as a function of n, the number of items to be sorted). Counting comparisons, rather than all low-level machine operations, gives us a clean measure of efficiency that does not depend on details of the machine, the programming language, or the implementation. For the same reason, it is appropriate to drop constant factors (like the $\frac{1}{2}$ here) as well as lower order terms (like the $-\frac{1}{2}n$ here). We say that the running time of these methods is "$O(n^2)$" (pronounced "order n^2" or "big-Oh of n^2") to indicate that we are using n^2 as a crude measure here. (We'll talk more about O-notation later in the course.)

Is an $O(n^2)$ sorting algorithm any use? For example, suppose we need to sort 2.1 million brokerage accounts by account number using a 1GHz PC. Let us say, optimistically, that we average 1 comparison every 10 CPU cycles (including all the loading, storing, and list manipulations). Then we can do 100 million comparisons per second. An algorithm using $\frac{1}{2}n^2$ comparisons takes over 6 hours. This seems like a lot.

Can we come up with a sorting algorithm that is faster than $O(n^2)$? Presumably we couldn't possibly beat $O(n)$, since it takes n steps just to look at all n items. It turns out that we can almost achieve this ideal behavior, with a clever use of recursion. We will get a sorting algorithm, called "Mergesort", which requires only $O(n\log_2 n)$ comparisons to sort n items. For the problem described in the previous paragraph, this amounts to 0.4 seconds.

This use of recursion is one of the simplest examples of the powerful technique known as "divide and conquer". You will see many more examples in later courses. The idea is to *divide* the input into two or more smaller pieces, or "subproblems", each of which can then be "conquered" by a recursive application of the same algorithm. Finally, the solutions to the subproblems need to be glued together appropriately to form a solution to the original problem. Divide and conquer algorithms vary greatly in the sophistication of the "dividing" and "gluing" operations.

For the sorting problem, a natural way to divide the input is to simply chop the list of n items into two lists each of size $\frac{n}{2}$; we can do this by simply taking the first $\frac{n}{2}$ items and the remaining $\frac{n}{2}$ items. (For simplicity, to avoid writing lots of floors and ceilings, we will assume that n is a power of 2, so that such even splits will always be possible. This assumption is not necessary in practice.) Then we recursively sort each of the two smaller lists. Finally, we "merge" the sorted sublists together to obtain a sorted version of the original list. The algorithm is written out in a bit more detail below:

```
algorithm Mergesort($S$)
$\{$S is a list of n items from a totally ordered set, n is a power of two$\}$
if $|S| = 1$ then return $S$
else
  divide $S$ into $T$ (the first $\frac{n}{2}$ items) and $U$ (the remaining $\frac{n}{2}$ items)
  $T' :=$ Mergesort$(T)$
  $U' :=$ Mergesort$(U)$
  $S' :=$ Merge$(T', U')$
  return $S'$
```

This algorithm is not yet completely specified as we have not defined the procedure "Merge". However, the idea here should be obvious: to merge two sorted lists, T and U, into a single sorted list, we scan through both lists once "in parallel", at each step placing the smaller of the two items we are viewing into the output list. Thus we begin by comparing the first items, t_1 and u_1, and placing the smaller of them (assume it is u_1) as the first item of the output list. Next we compare t_1 and u_2 and again place the smaller one (assume it is t_1) as the second item of the output list. Next we compare t_2 and u_2, and place the smaller of the two as the third item of the output list, and so on until one list is exhausted. At this point we finish up by appending

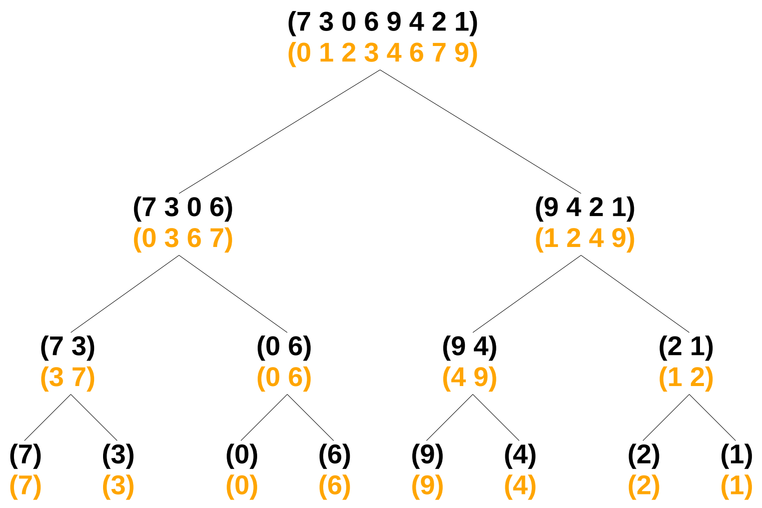

Figure 1: Tree of recursive calls to Mergesort, showing input and output lists.

the remaining elements of the remaining list to the end of the output list. This procedure requires at most one comparison for each element of the merged list (except the last), so the total number of comparisons required to merge two lists of combined length n is at most $n-1$.

INPUT–OUTPUT SPECIFICATION

We will leave the writing of Merge as an exercise. We will, however, give its **input–output specification**, i..e, the properties required to hold of its output, given suitable inputs. The proof is left as an exercise.

Lemma 5.1: *Merge If A and B are disjoint, sorted lists, then* $\text{Merge}(A,B)$ *is a sorted permutation of $A \cup B$.*

Now we will prove that Mergesort is correct. Notice that we can do this *even though we have not written the Merge subroutine*, because we only need the *specification* of Merge to complete the proof for Mergesort. Similarly, to analyze the runtime of Mergesort, we need to know just the runtime of Merge (which does *not* follow from its specification).

Theorem 5.1: *The algorithm Mergesort, when presented with a list S of n distinct items (n a power of two), outputs a sorted version of S.*

Proof: First, we should state the theorem a bit more precisely. What exactly do we mean by “a sorted version of S”? We mean that the output list S' has the following two properties:

(i) The list S' is a *permutation* of the list S (i.e., it consists of precisely the same items, possibly in some other order).

(ii) The list S' is *sorted*, i.e., if $S' = [s'_1, s'_2, \ldots, s'_n]$ then $s'_1 < s'_2 < \ldots < s'_n$.

Let’s now prove by induction on the length n of the input list S that the output list S' is a sorted version of S. The base case ($n = 1$) is easy: here $S' = S$, which is indeed a sorted version of itself because it consists of only one element.

Now consider the case $n > 1$ (n a power of two). In this case, from the definition of the algorithm, we see that the output S' is precisely $\text{Merge}(T', U')$, where $T' = \text{Mergesort}(T)$ and $U' = \text{Mergesort}(U)$, and T, U are the first and second halves of S respectively. Since T and U are each lists of length only $\frac{n}{2}$, we can apply the inductive hypothesis to deduce that T' and U' are sorted versions of T and U respectively; i.e., both T' and U' satisfy properties (i) and (ii) above.

To complete the proof, we now need to appeal to the properties of Merge. Because T' and U' are sorted, disjoint lists, the Merge lemma tells us that S' is a sorted permutation of $T' \cup U'$. Now we appeal to the following general property of permutations: for any disjoint lists T, U, if T' is a permutation of T and U' is a permutation of U, then $T' \cup U'$ is a permutation of $T \cup U$. Now $T \cup U$ is a permutation of S by definition. Hence S' is a sorted permutation of S. $\square$

Let's now turn to the efficiency of Mergesort. As before, we'll just count the number of comparisons required to sort n items. We can give an informal argument by examining the recursion tree shown in Figure 1. The first level (root) consists of one problem of size n, the second level of two problems of size $\frac{n}{2}$, the third of four problems of size $\frac{n}{4}$, and so on. The total number of levels is $\log n + 1$, with the bottom (leaf) level consisting of n problems of size 1. How many comparisons are performed at each level? Well, at the root we perform $n-1$ comparisons in the worst case. At the second level, we perform (in the worst case) $\frac{n}{2} - 1$ comparisons for each of the two subproblems, for a total of $n-2$. At the third level we perform $4 \times (\frac{n}{4} - 1) = n - 4$ comparisons, and so on. Thus at each level we perform at most n comparisons (except for the leaves, where there are no comparisons). Since there are $\log n$ non-leaf levels, the total number of comparisons is at most $n \log n$.

If we count more carefully, we can establish the following upper bound:

Theorem 5.2: *The maximum number of comparisons, $C(n)$, performed by the algorithm Mergesort for an input list of length n (a power of 2) satisfies $C(n) \le n \log n - n + 1$.*[1]

Proof: We prove the claim by (strong) induction on n. The base case ($n = 1$) is easy: here Mergesort performs no comparisons (it simply outputs the original list), and the value of $n \log n - n + 1$ when $n = 1$ is indeed 0. So the base case holds.

Now consider the case $n > 1$ (n a power of two). Mergesort performs two recursive calls, each on a list of length $\frac{n}{2}$, and calls Merge on two lists of combined length n. Hence

$$C(n) \le 2C(\frac{n}{2}) + (n-1).$$

This is the crucial step: make sure you understand it! The first term on the right comes from the two recursive calls (each of which, by definition of $C(\cdot)$, requires at most $C(\frac{n}{2})$ comparisons); the second term comes from the Merge step (which requires at most $n-1$ comparisons).

Now we can complete the proof with a little algebra, together of course with an application of the inductive hypothesis:

$$\begin{aligned} C(n) &\le 2C(\tfrac{n}{2}) + n - 1 \\ &\le 2\left\{\tfrac{n}{2}\log(\tfrac{n}{2}) - \tfrac{n}{2} + 1\right\} + n - 1 \\ &= n(\log n - 1) + 1 \\ &= n \log n - n + 1. \end{aligned}$$

In the second line here we used the (strong) induction hypothesis. The theorem is thus verified by induction. $\square$

[1] Unless otherwise stated, "log" denotes base-2 logarithm.